Interface Generation and Compositional Verification in JavaPathfinder

Dimitra Giannakopoulou and Corina Pasareanu

NASA Ames Research Center,
Moffett Field, CA 94035, USA,
{dimitra.giannakopoulou, corina.s.pasareanu}@nasa.gov

Abstract. We present a novel algorithm for interface generation of software components. Given a component, our algorithm uses learning techniques to compute a permissive interface representing legal usage of the component. Unlike our previous work, this algorithm does not require knowledge about the component's environment. Furthermore, in contrast to other related approaches, our algorithm computes permissive interfaces even in the presence of non-determinism in the component. Our algorithm is implemented in the JavaPathfinder model checking framework for UML statechart components. We have also added support for automated assume-guarantee style compositional verification in JavaPathfinder, using component interfaces. We report on the application of the presented approach to the generation of interfaces for flight software components.

1 Introduction

Component interfaces are a central concept in component-based software engineering. Although in current practice, interfaces typically describe the services that a component provides and requires at a purely syntactic level, the need has been identified for interfaces that document richer aspects of component behavior. Such extended interfaces are usually not provided, which makes their automatic generation an area of active research[?,?,?].

This paper addresses the automatic generation of interfaces that describe legal sequences of component calls. Such interfaces can serve as a documentation aid to application programmers, but can also be used by verification tools in checking that components are invoked correctly within a system. In fact, component interfaces are key for modular program analysis. They reduce the task of verifying a system consisting of a component and a client, to the more tractable task of verifying that the client satisfies the component's interface.

In previous work [?,?,?], we presented a framework based on learning, to perform automated assume-guarantee model checking of safety properties. To check that a system consisting of components M_1 and M_2 satisfies a safety property P, the framework automatically builds and refines *assumptions* A for one of the components, for example M_1, to satisfy P, which it then tries to discharge on

the other component, M_2. Although assumptions A essentially constitute interfaces for component M_1, their generation relies on knowledge of component M_2. Moreover, the focus of the framework was to compute assumptions that would allow to prove or disprove the property in the system, rather than assumptions that precisely document the behavior of a component.

The algorithm presented here for interface generation is also based on learning. However, in contrast to our work discussed above, it concentrates on the creation of precise component interfaces, *irrespective* of the component clients. By precise, we mean *safe* and *permissive*, as defined in [?]. An interface is safe if it accepts no illegal sequence of calls to the component. An interface is permissive if it includes all the legal sequences of calls to the component. Moreover, in [?], we presented an algorithm for generating what we call *weakest* assumptions in the context of Labeled Transition Systems. Weakest assumptions essentially constitute precise component interfaces. The difference of the current algorithm is that it is iterative, meaning that it can return partial results. Moreover, our experience has been that the learning-based approach is more efficient for components that have relatively small interfaces.

Henzinger et al. also target the generation of safe and permissive interfaces in [?]. Unlike our framework, their work based on abstraction techniques and it is only applicable to components that are *visibly deterministic*. The latter requires that the behavior of the component be deterministic with respect to the methods / actions in its communication interface (we will henceforth call the communication interface of a component its *alphabet* in order to avoid confusion with interface in this context). In the applications that we have been dealing with, this requirement proved too restrictive. For example, we often need to generate interfaces that focus on specific aspects of the component behavior, and that therefore include only a subset of the component's alphabet. Components that are visibly deterministic with respect to their full alphabet, typically lose this property when a subset of that alphabet is considered. Finally, Alur et al. also use learning to synthesize interface specifications for Java Classes. However, their approach is heuristic-based, meaning that they do not always obtain precise interfaces.

We have implemented our algorithms in the JavaPathfinder (JPF) model checking framework for UML statechart components[?]. We have also added support for automated assume-guarantee style compositional verification in JPF, using component interfaces. JPF is an open source model checker for Java programs which, until now, provided no support for compositional verification. This work, which is included in a new extension (namely *CV*), adds the following features to JPF: 1) support for verification of safety properties expressed as finite state automata; 2) support for learning-based interface generation; and 3) support for assume-guarantee reasoning, where assumptions and guarantees are both expressed as finite-state automata. We finally report on the application of the presented approach to the generation of interfaces for flight software components.

The contributions of this work can be summarized as follows:

1. A novel algorithm for automated generation of precise component interfaces, also applicable to components that are not visibly deterministic
2. Implementation of our algorithm in the JPF open source model checker. In addition to interface generation, we have provided support for assume-guarantee reasoning in JPF, where assumptions and guarantees are both expressed as finite-state automata.
3. Case studies in the context of NASA applications that demonstrate the use of our algorithm in practice.

Related Work The work closest to ours was discussed above. Several other approaches to automatic generation of component interfaces have been proposed in the literature. For example, Whaley et al. [] use a combination of static and dynamic analyses to generate interfaces for Java components. Tkachuk et. al [] use static analysis to obtain component abstractions, used as environments during modular analysis. Some approaches are based on extracting interfaces from sample execution traces [?,?]. All these techniques generate approximate interfaces, as opposed to our work that aims at producing interfaces that provide correctness guarantees. Interface generation is related to compositional verification. In particular, assume-guarantee reasoning is a compositional approach that uses assumptions when reasoning about components in isolation [?,?,?,?]. Component interfaces can be used as assumptions in this context.

2 Background

We model software components using labeled finite state transition systems (LTSs), where transitions are labeled with component actions.

Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment. Let π denote a special *error state*, which models safety violations in the associated transition system; π has no outgoing transitions.

LTSs An LTS M is a four-tuple $\langle Q, \alpha M, \delta, q_0 \rangle$ where: Q is a finite non-empty set of states; $\alpha M \subseteq \mathcal{A}ct$ is a set of observable actions called the *alphabet* of M; $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is a transition relation; and $q_0 \in Q$ is the initial state.

Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q_0' \rangle$. M *transits* into M' with action a, denoted $M \xrightarrow{a} M'$, if $(q_0, a, q_0') \in \delta$ and either $Q = Q'$, $\alpha M = \alpha M'$, and $\delta = \delta'$ for $q_0' \neq \pi$.

An LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is *non-deterministic* if it contains τ-transitions or if there exists $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic*.

Traces A *trace* t of an LTS M is a finite sequence of observable actions that label the transitions that M can perform starting at its initial state, ignoring the τ-transitions. For $\Sigma \subseteq \mathcal{A}ct$, we use $t \uparrow \Sigma$ to denote the trace obtained

by removing from t all occurrences of actions $a \notin \Sigma$. For a set of traces T, $T \uparrow \Sigma = \{t | \exists t' \in T.t' \uparrow \Sigma = t\}$.

Parallel Composition Parallel composition "$\|$" is a commutative and associative operator such that: given LTSs $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$, $M_1 \| M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows (the symmetric version also applies): (1) $M_1 \| M_2 \xrightarrow{a} M_1' \| M_2$ if $M_1 \xrightarrow{a} M_1'$ and $a \notin \alpha M_2$, and (2) $M_1 \| M_2 \xrightarrow{a} M_1' \| M_2'$ if $M_1 \xrightarrow{a} M_1'$, $M_2 \xrightarrow{a} M_2'$, and $a \neq \tau$.

3 Interface Generation

In this section we define safe and permissive interfaces for software components and we describe our approach to synthesizing such interfaces autmatically.

3.1 Safe and Permissive Interfaces

Let M be a software component. For simplicity of presentation, we will first assume that M includes an error state that expresses the undesired behavior of M (for example, some asertion violations). This same case is considered in the related work of Alur et al. [] and Henzinger et al. []. Later in this section we will discuss the more general case where the component property is given as a separate (safety) automaton.

Let $\Sigma \subseteq \alpha M$ denote the communication alphabet of component M, i.e., the set of actions through which M communicates with its environment. Our goal is to compute M's precise interface as a finite state automaton A over Σ. As mentioned, we need to make sure that A is both *safe* and *permissive*, as defined formally below.

Let us first define the legal and illegal languages of component M. A word $t \in \alpha M^*$ is *illegal* if it corresponds to *some* trace of M that leads to error state π; otherwise, the word is *legal*. Then $\mathcal{L}_{legal}(M)$ denotes the set of legal words of M and $\mathcal{L}_{illegal}(M)$ denotes the set of illegal words of M. Note that $\mathcal{L}_{legal}(M)$ and $\mathcal{L}_{illegal}(M)$ are complementary. Furthermore, note that, while illegal words correspond to actual traces in the component, legal words may also represent behavior that is never executed by the component (and hence could never lead to violations).

Definition 1. *A is a* safe *interface iff* $\mathcal{L}_{legal}(A) \cap \mathcal{L}_{illegal}(M) \uparrow \Sigma = \emptyset$.

In other words, an interface is safe if it accepts no illegal words of M.

Definition 2. *A is a* permissive *interface iff* $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$.

In other words, an interface is permissive if it accepts all legal words of M.

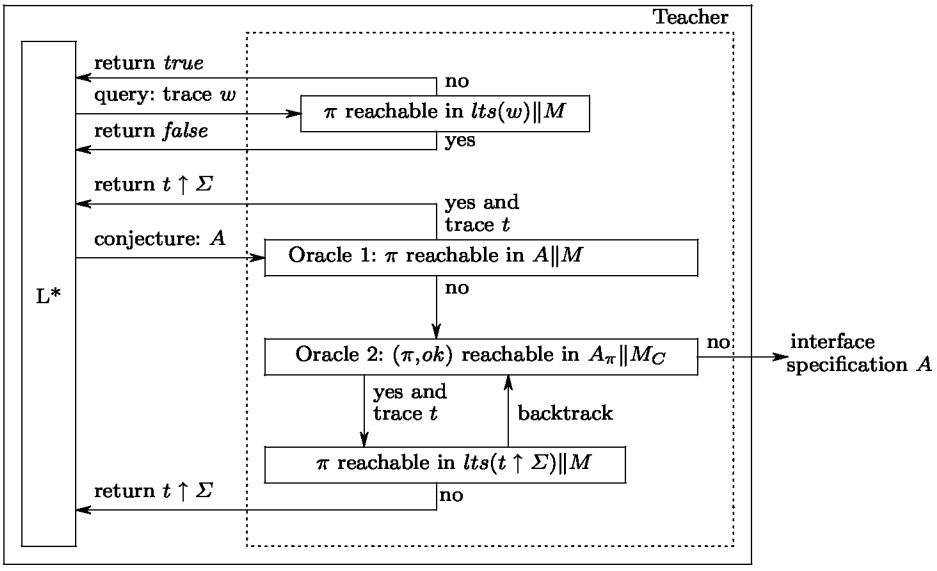

Fig. 1. Learning interface specifications with L*

3.2 Learning Interface Specifications with L*

Our approach for learning interface specifications is illustrated in Figure 1. We use an off-the-shelf learning algorithm, namely L* [**?**], to iteratively compute the interface specification A for M that is both *safe* and *permissive*. L* learns an unknown language (over a given alphabet) and produces a deterministic finite state automaton that accepts it; the learning process is iterative and it uses a *teacher* that provides answers to queries and counterexamples to conjectures (for more details on L* see []). In our framework, the problem of answering queries and counterexamples is reduced to reachability problems, solved by a model checker.

Queries L* is first used to repeatedly *query* M to check whether or not, in the context of strings w, M violates the property. This is equivalent with checking if an error state π is reachable in $lts(w)\|M$. Here $lts(w)$ denotes an LTS over Σ that accepts only string w. The results of the queries are used by L* to first make a "conjecture", i.e. it builds an automaton A that accepts all the strings for the positive queries (the case error unreachable), and does not accept the strings for the negative queries (the case error reachable).

The conjectured automaton A is then checked to make sure it is both safe and permissive. This is done with the help of a *teacher* that implements two oracles as described below.

Oracle 1 checks if A is *safe* by checking whether π is reachable in $A \parallel M$. If it is, then it means that A is un-safe. The resulting counterexample t, projected on the interface alphabet Σ, is returned to L* to refine its conjecture. If the error state is un-reachable, then it means A is safe and we proceed to Oracle 2.

Oracle 2 checks if safe interface A is also permissive, i.e. we want to check that $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$. This amounts to making sure that there are no words $w \in \Sigma^*$ such that $w \in \mathcal{L}_{legal}(M) \uparrow \Sigma \cap \mathcal{L}_{illegal}(A)$. This is equivalent to $w \in \mathcal{L}_{illegal}(A)$ and $\forall t \in \alpha M$ such that $w = t \uparrow \Sigma$, $t \in \mathcal{L}_{legal}(M)$.

We search for such words using a special reachability procedure performed on $A_\pi \parallel M_C$ (see pseudo-code in Figure 2). Here A_π denotes the *completion* of A with an error state, i.e. we complete each state with outgoing transitions to π, such that each state has outgoing transitions labeled with every action in Σ. Similarly, M_C denotes the *completion* of M with a special *s*ink state. We need these constructions to reason about traces in $\mathcal{L}_{illegal}(A)$ and $\mathcal{L}_{legal}(M)$, respectively. Note that $\mathcal{L}_{illegal}(A) = \mathcal{L}_{illegal}(A_\pi)$ and $\mathcal{L}_{legal}(M) = \mathcal{L}_{legal}(M_C)$. Note that, for Oracle 2, since both A_π and M_C contain error states, we need to distinguish between the two in $A_\pi \parallel M_C$ (this was not necessary for queries and Oracle 1).

Given the above constructions, checking permissiveness reduces to checking reachability of states of the form: (π, ok), were π is an error state coming from A_π and ok denotes a non-error state in M_C. If such a combined state is found, then the trace t leading to it *may* indicate that A is not permissive, since $w = t \uparrow \Sigma$ leads to an error state in A_π but it is legal in M_C (and hence in M). However, due to non-determinism in M (and hence in M_C), it may be the case that on another path, t *does* lead to the error state. Even if this is not the case, there may exist other traces t' such that $w = t' \uparrow \Sigma$ and t' leads to an error in M_C on a different path (see Figure 3). We check both these cases by performing a *query* on $t \uparrow \Sigma$. Note that *we do not stop the state space exploration in JPF*, but rather, we take trace t that is returned, and we check if, in the context of $t \uparrow \Sigma$, M violates its properties.

If the query returns true, then it means the interface is not permissive, and therefore $t \uparrow \Sigma$ is returned to L* for refinement, and the learning process continues with more queries and eventually with a new conjecture.

If the query returns false, then t does not correspond to a real counterexample. Model checking therefore ignores this state; it backtracks, and then continues its state space exploration. If no traces that satisfy the condition above exist, then indeed the conjectured automaton is also the most permissive interface, and therefore it is output to the user.

We note that every query is stored in the L* memoized table, so the result of the query on the same trace $t \uparrow \Sigma$ later (when A is the same) will be obtained directly (and faster) from the table.

Example: dealing with observable non-determinism We use the Example in Figure 3 to illustrate the reason for the extra query to deal with non-

Oracle 2
input: safe interface A;
begin
(1) Model-check $A_\pi \| M_C$:
(2) **if** $(\pi,\ ok)$ is reachable by trace t **then**
(3) **if** π is not reachable in $lts(t \uparrow \Sigma) \| M$ **then**
(4) **return** $t \uparrow \Sigma$ to L*;
(5) **else**
(6) **backtrack**;
(7) **output:** safe and permissive interface A;
end.

Fig. 2. Oracle 2

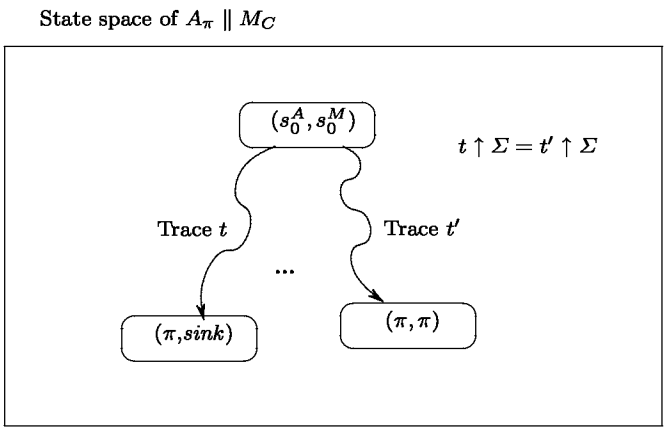

Fig. 3. Example for Oracle 2: dealing with non-determinism

determinism in the component behavior. Assume that model checking $A_\pi \| M_C$ finds a state of the form $(\pi,\ ok)$ that is reachable by trace t.

Assume also that there is another trace t' such that $t \uparrow \Sigma = t' \uparrow \Sigma$; we say that M (and M_C) is *observable non-deterministic* (to keep a similar terminology to the one introduced in []).

Furthermore, assume that t' leads to (π, π). In this case, A is permissive with respect to w; therefore, the model-checker is instructed to backtrack and then to continue the search for other states of the form $(\pi,\ ok)$ (that may indicate that A is not permissive enough).

Properties as safety automata Assume now that M does not have error states, and we want to generate an interface specification for ensuring a property P, given as a (deterministic) safety automaton, encoding all the desired behaviors of the component. Conversely, P_π encodes all the un-desired behaviors of the

component. The procedure described above will be exactly applicable to this case as well, if we treat $M||P_\pi$ as M above.

3.3 Correctness and Termination

We argue here the correctness and termination of our approach. To argue correctness, we first show that Oracle 1 guarantees a safe interface while Oracle 2 guarantees a permissive interface.

Proposition 1. *π is un-reachable in $A \parallel M$ iff $\mathcal{L}_{legal}(A) \cap \mathcal{L}_{illegal}(M) \uparrow \Sigma = \emptyset$.*

Proof. "If:" By contradiction. Assume $\mathcal{L}_{legal}(A) \cap \mathcal{L}_{illegal}(M) \uparrow \Sigma \neq \emptyset$. Then, $\exists t \in \alpha M^*$ such that $t \uparrow \Sigma \in \mathcal{L}(A)$ and t leads to error in M. Then t is also leading to an error in $A \| M$ which is a contradiction. q.e.d.

"Only If:"

Proposition 2. *$(\pi, \mathrm{o}k)$ is un-reachable in $A_\pi \| M_C$ iff $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$.*

Proof. "If:" By contradiction. Assume $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$ does not hold. Then $\exists t \in \alpha M^*$ such that $t \in \mathcal{L}_{legal}(M)$ and $t \uparrow \Sigma \in \mathcal{L}_{illegal}(A)$. Then t is also leading to $(\pi, \mathrm{o}k)$ in $A_\pi \| M_C$ which is a contradiction. q.e.d.

"Only If:"

Theorem 1. *Given component finite state M (that may include error states), the algorithm implemented by our approach terminates and it returns a safe and permissive interface A.*

Proof. Correctness follows from the two propositions above. Termination follows from the correctness of L^*, which is guaranteed that, if it keeps receiving counterexamples, it will eventually terminate.

We note here that the approach of Henzinger et al. [] can only handle components that are observable deterministic. However, we believe that this is very limited in practice, since even if the component itself is deterministic, considering only its interface behavior (and abstracting away its internal behavior) leads to non-determinism.

Furthermore, the approach of Alur et al. ?? proposes to generate safe interface specifications for Java components, that are made finite state using predicate abstraction techniques (and therefore they can be non-deterministic). However the approach does not guarantee that the interface is also permissive, since it only uses heuristics to implement what it amounts to Oracle 2 (which is called "superset query" in []). That work argues that Oracle 2 can not be implemented efficiently, since it involves determinization of component M. We note that in our approach we avoid the expensive determinization step by performing the extra query on the traces that lead to (π, ok). While the worst case complexity of our approach can not be avoided, in practice we noticed that Oracle 2 is not called very often.

How does it compare with our old approach at ASE? It also involved determinization.

4 Compositional verification in JPF

4.1 Java PathFinder

Java PathFinder (JPF) [?] is a verification framework developed by the RSE group at NASA Ames. It has been started as an explicit state model checker for Java byte-code. The focus of JPF is on finding bugs, such as concurrency related bugs (deadlocks, races, missed signals etc.), runtime related bugs (e.g. unhandled exceptions), etc. JPF can also check for violations of user-specified assertions that encode application specific requirements. JPF uses a variety of scalability enhancing mechanisms, such as user extensible state abstraction and matching, on-the-fly partial order reduction, configurable search strategies, and user definable heuristics (searches, choice generators). JPF has been open sourced since 04/2005 under NOSA 1.3 license and it is available for download from javapathfinder.sourceforge.net.

4.2 JPF's UML Statechart Extension

To address the needs for model analysis of flight software, JPF has recently been extended with a state-chart modeling and analysis capability that allows Java modeling of UML state machines []. Many UML development systems can produce code from diagrams, but this code is usually aimed at production systems, and is not suitable for software model checkers. The approach taken in JPF (Figure 4) is based on a specific translation scheme from UML state charts into Java code that (a) is highly readable, (b) shows close correspondence between diagram and program, (c) provides a 1:1 mapping between model and program states, and (d) imposes no restrictions about aspects and actions that can be modeled.

The JPF statechart extension is specialized to handle the obtained Java models more efficiently than random Java code. These Java models can be run in isolation, which corresponds to running them in the context of an external environment that may provide any input event at any stage (we will call this the universal environment). Alternatively, a guidance script may be provided by the user, which represents the input event sequences that can be provided by the external environment.

We have used the JPF statechart extension to implement our interface synthesis algorithms for components expressed in the JPF statechart framework. In the context of this work, we do not attempt to perform compositional reasoning for UML statecharts. The reason is that statechart composition semantics is obfuscated and setting up compositional reasoning for statecharts is a challenge even at a purely theoretical level. Rather, we use UML statecharts, as supported by JPF, to represent finite state components with Labeled Transition System semantics. Therefore composition of components comes down to LTS composition, as described in Section ??. The interfaces that we generate are expressed as LTSs in the FSP notation [].

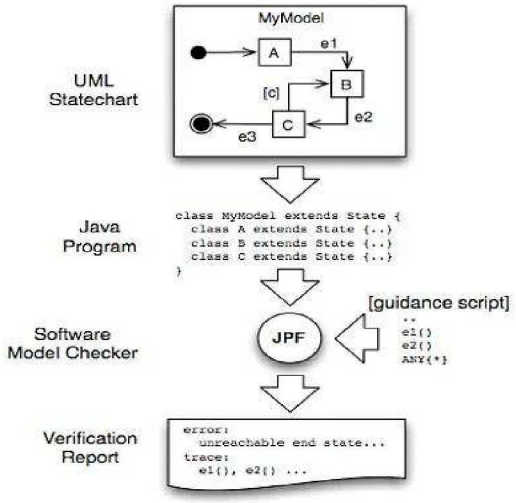

Fig. 4. Example illustrating JPF's UML extension

4.3 Assume-guarantee Reasoning in JPF

Model checking using assumptions and properties has been implemented using JPF listeners. A JPF listener is an extension mechanism that enables client code to be informed of certain events that occur while JPF performs its search. Listeners can also interact with JPF, for example, a property listener will inform JPF of the fact that a property has failed, through a condition that it provides in its check() method. Both assumptions and properties are implemented with the `gov.nasa.jpf.cv.SCSafetyListener` class, which extends the gov.nasa.jpf.PropertyListenerAdapter class, provided by JPF to ease property creation. On creation, a `SCSafetyListener` is associated with a finite state automaton (`gov.nasa.jpf.cv.SCSafetyAutomaton`) P, which expresses the property or assumption to be used during model checking. Note that the state of a listener is not included in the state that JPF explores / stores during model checking. However, the state of the automaton P needs to be part of the state space for correct state-space exploration and backtracking. *** We perform this by adding a static integer field AssumptionState of class CVState for the CV extension. It can be set as follows:

```
MJIEnv env = ti.getEnv();
env.setStaticIntField("gov.nasa.jpf.cv.CVState", "AssumptionState",
P.getCurrentState());
```

An `SCSafetyListener` listens for and reacts to the following events:

- instructionExecuted: signals the fact that an instruction was executed by JPF. The reaction of the listener is to invoke method `advance(...)` on

the automaton P. Advancing the automaton corresponds to making a state transition, if the instruction that was executed corresponds to an action in the alphabet of the automaton. If a transition on an alphabet action is undefined from the current state, this is an illegal transition (corresponds to a transition to the *error* state). For properties, this means that an error has occurred, so the result returned by the check() method of the listener is set to false.

- choiceGeneratorAdvanced: signals the fact that the next statechart action is selected for execution. The reaction of the listener is to enquire with P whether this action would make it transition to the error state if it were to be executed (this does not change the state of P since the transition is not really executed yet). Reaching an error state in an assumptions means that the current path explored is not a valid path under this assumption and must therefore be ignored. The listener performs this as follows: `vm.getSystemState().setIgnored(true)`, which requests for JPF to backtrack.
- stateBacktracked: when the model checker backtracks, then the automaton must backtrack accordingly. We perform this by getting the JPF path to the current model state after backtracking, and replay the path on the automaton, to ensure that the two are in synch.
  Could we not do getStaticIntField instead???

For example, in order to check some property described as an automaton provided in some file `Foo`, we need to include the following arguments when running JPF's main class `gov.nasa.jpf.JPF`:

```
+jpf.listener=.cv.SCSafetyListener
+safetyListener1.property= Foo
```

The first argument informs JPF that an `SCSafetyListener` will need to be notified of specific events, and the second one provides details for the listener, i.e., its unique id is "1", it is of type property (as opposed to assumption), and the automaton associated with it is provided in file `Foo` (this may also include the full path to `Foo`).

4.4 Interface Generation and Discharge

Interface generation in JPF can be performed by invoking the main class `gov.nasa.jpf.tools.cv.ScRunCV`. Argument `+assumption.alphabet=<actions>` is used to define the alphabet of the interface to be generated, in terms of method names. Argument `+assumption.outputFile=<file name>` defines a file in which the generated interface is output. This allows for a generated interface to be used for subsequent reasoning, either as an assumption, or as a property. The format currently used for expressing the interface is the FSP language [].

The main method of `gov.nasa.jpf.tools.cv.ScRunCV` creates an instance of class `gov.nasa.jpf.tools.cv.SETLearner` to carry out the learning of the interface, with an associated instance of `gov.nasa.jpf.tools.cv.SCModularTeacher` to serve as the teacher. Our learning algorithm implementation uses JPF to

```
public boolean query(Vector sequence) throws SETException {

  Boolean recalled = memoized_.getResult(sequence);
  if (recalled != null) {
      return (!recalled.booleanValue());
  } else {
      // play the query as an assumption
      System.out.println("\n New query: " + sequence);
      SCSafetyListener assumption = new SCSafetyListener(
          new SCSafetyAutomaton
                  (true, sequence, alphabet_, "Query", module1_));

      JPF jpf = createJPFInstance(assumption, property, module1_);
      jpf.run();
      boolean violating = jpf.foundErrors();
      memoized_.setResult(sequence, violating);
      return (!violating);
  }
}
```

Fig. 5. Answering queries in `SCModularTeacher`

perform the model checking steps described in Section ??. JPF model checks individual components in the context of the universal environment. Listeners are added as necessary to reflect the work of the Teacher, which consists of answering Queries, and implementing Oracle 1 and Oracle 2 in order to answer conjectures, as described in more detail below.

Queries and Oracle 1. Queries and Oracle 1 are performed in a similar fashion because they are concerned with checking whether error states are reachable in the component, in the context of a particular sequence (for queries) or finite state automaton (for Oracle1). As illustrated in Figure 5, to respond to a query, a listener instance `assumption` is created with an associated automaton that reflects the particular sequence that is being queried. The automaton is considered as an assumption. JPF is then invoked, together with the `assumption` listener. If JPF returns errors, the answer to the query is `false`, otherwise the answer is `true`. Oracle 1 works in a similar fashion, with the difference that it also returns a counterexample.

Oracle 2. Oracle 2 checks for permissiveness of a computed interface. It needs to work on the completed component, as described in Section ??. This is a manual step that we intend to automate in the future. It similarly invokes JPF, but performs the search in the context of a specialized type of listener, the `gov.nasa.jpf.cv.SCConformanceListener`. Its aim is to detect the reachability of a $(ok, error)$ combination of states in the component and interface where the component is in a non-error state, while the interface is in an error state.

The `gov.nasa.jpf.cv.SCConformanceListener` listens for and reacts to the following events:

- executeInstruction: When the instruction about to be executed by JPF is an assertion violation, then it means that the component has entered an error state. Since such states are not targeted by the listener, it performs `ti.skipInstruction();` followed by `vm.getSystemState().setIgnored(true);`. The first command ensures that the exception is not processed by JPF, for efficiency. The second asks JPF to backtrack since this path cannot lead to the targeted combination of states. Note that choiceGeneratorAdvanced (which `gov.nasa.jpf.cv.SCSafetyListener` listens to), does not monitor exceptions.
- instructionExecuted: reacts similarly to `gov.nasa.jpf.cv.SCSafetyListener`. When the automaton associated with the listener moves to an error state, the result returned by the check() method of the listener is set to false. This is because the component is in a legal state (illegal states are never reached since the listener advises JPF to backtrack when it reacts to executeInstruction events), while the interface is in an error state.
- stateBacktracked: reacts similarly to `gov.nasa.jpf.cv.SCSafetyListener`.

As described in Section ??, when an $(ok, error)$ state is detected by the model checker, if the counterexample leading to this state is queried, so that if it is not a real counterexample, the model checker will backtrack. Since a query involved calling the model checker, this would involve nested model checker calls. To avoid such nesting, our implementation exploits a memoized table that is used by the learner to store results of previous queries. Oracle 2 checks for the reachability of $(ok, error)$ states in a loop. Whenever a counterexample is obtained by the model checker, then Oracle2 invokes a query on it. Each query stores its result in the memoized table.

Whenever a real counterexample is obtained, then Oracle 2 exits the loop and reports the result to the learner. When a counterexample is spurious, then another iteration of the loop is entered. In this iteration, we wish to ensure that the model checker will not report the same spurious counterexample. We achieve this as follows. When a `gov.nasa.jpf.cv.SCSafetyAutomaton` is asked to advance in the context of a `gov.nasa.jpf.cv.SCConformanceListener`, if the automaton reaches an error state, it will get the path to this state from JPF. It will then check the memoized table to see if there is a result for the corresponding sequence stored there. If there is, and the result is true, then it means that this is a spurious counterexample, and it notifies JPF to backtrack. Therefore, we have implemented the nested model checking calls by consecutive calls to the model checker, where the information of spurious counterexamples is shared through the memoized table.

Interface discharge. For compositional reasoning, one needs to also discharge the generated interface on the component environment. This can be performed by model checking the environment in the presence of a `gov.nasa.jpf.cv.SCSafetyListener` using the interface as a property.

5 Experimental results

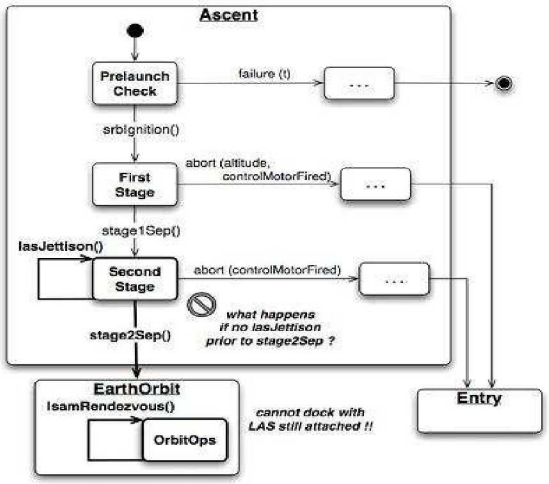

Fig. 6. Model of the Ascent and Earth Orbit flight phases of a spacecraft

In order to evaluate our implementation, we used a state-chart model of the *Ascent* and *EarthOrbit* flight phases of a space-craft (see Figure 6). The JAVA model is available with the JPF distribution under `examples/jpfESAS`. The UML statechart diagrams corresponding to the model are included in `examples/jpfESAS.doc`.

The model was created and used to demonstrate the features of the JPF UML statechart extension to our mission customers. Several properties were analyzed on the model, and JPF returned violations for some of these properties. When the counterexamples obtained were analyzed, it was clear that some of the violations were spurious. The violations were related to the following properties:

- An event *lsamRendezvous*, which represents a docking maneuver with another spacecraft, fails if the LAS (launch abort system) is still attached to the spacecraft.
- Event *tliBurn* (trans-lunar interface burn takes spacecraft out of the earth orbit and gets it into transition to the moon) can only be invoked if EDS (Earth Departure Stage) rocket is available.

These violations were due to the fact that the universal environment was too general. The models had been created under the assumption that the use of the model respects some implicit flight rules. We decided to use our interface generation techniques to formalize the flight rules. More specifically, for each property, we generated a safe and permissive interface to eliminate its corresponding violations. To do this, we added a listener that eliminated all assertion violations that were not related to the targeted property, through the following arguments:

```
+jpf.listener=.tools.ChoiceTracker:.cv.AssertionFilteringListener
+assertionFilter.include=<method_name>
```

These arguments specify that all assertion violations that occur outside the particular `method_name>` will be ignored.

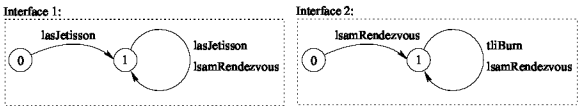

Fig. 7. Generated interface specifications encode assumptions about component environments

The generated interface specifications are illustrated in Figure 7. The first one expresses the fact that the *lsamRendezvous* maneuvers cannot start before the *las* module of the spacecraft has jettisoned. According to the second one, it does not make sense to perform *tliBurn* prior to performing *lsamRendezvous*. These interfaces were inspected by the developer of the model that confirmed that they encode actual flight rules. Interface generation can therefore be used by developers to help them in the expression of the assumptions that their models encode.

6 Conclusions

We have proposed an algorithm for automatically synthesizing behavioral interface specifications for finite state software components. Our algorithm is the first to compute precise interfaces even in the presence of non-determinism in the visible behavior of a component. We have implemented our approach in the JavaPathfinder model checking framework for UML statechart components, and have obtained promising results from its application to several systems. The source code of our implementation and the examples to which it has been applied are available through `javapathfinder.sourceforge.net`.

In the future, we plan to investigate interface generation for methods with parameters. We have made some initial experiments using JPF's symbolic execution extension to generate values for parameters with infinite domains, and used these values to define finite interface alphabets related to their corresponding methods. We wish to pursue this direction further, and also plan to generalize our results for generic Java components. For generic Java components that may be infinite, we will combine our approach with techniques such as predicate abstraction.